*To read the article online, visit http://www.4GuysFromRolla.com/articles/070109-1.aspx*

# Examining ASP.NET's Membership, Roles, and Profile - Part 15

*By Scott Mitchell*

## Introduction

When a visitor registers a new account on an ASP.NET website that uses the Membership system, they are prompted (by default) for their username, password, e-mail address, and other pertinent information. Along with functionality for registering new accounts, the ASP.NET Membership system provides page developers techniques for modifying information about users. For instance, with just a couple of lines of code you can change an existing user's e-mail address, approve a user, or unlock them (if their account was locked out). However, there are certain bits of user information that cannot be modified through the Membership API, such as the username.

For most sites this is a non-issue. Once a visitor has registered an account that username is fixed; if they want a different username, well, they'll just have to register a new account. But consider a website that has customized the account creation process so that instead of prompting the user for both a username and e-mail address, the user is only asked to enter an e-mail address and that it is used as both their username and e-mail address on file. Anytime a user switched e-mail addresses - which can happen when changing jobs, changing ISPs, or moving to the new, hip, web-based e-mail provider of the day - they need to also change their username on your site.

In order to change a user's username we'll need to bypass the Membership API and work directly with the user store. This article shows how to interface directly with the SQL Server database schema used by the `SqlMembershipProvider` to change an existing user's username. Read on to learn more!

## Modifying User Account Information

The .NET Framework includes two classes for programmatically working with user accounts via the Membership system. The first class is `Membership`, which has methods for creating, updating, and deleting user accounts along with methods for getting information about a particular user. The other class, `MembershipUser`, models a user in the Membership system and has properties that describe the user: `UserName`, `Email`, `IsApproved`, `LastLoginDate`, and so on. To change a user's e-mail address you would use these two classes in the following way:

1. Get information about the user via the `Membership.GetUser` method:

```
'Get information about the currently logged on user
Dim userInfo As MembershipUser = Membership.GetUser()
```

2. Set the various properties of the `MembershipUser` object returned by the `GetUser` method:

```
userInfo.Email = "newAddress@example.com"
```

3. Update the modified user account via the `Membership.UpdateUser(`*`MembershipUser`*`)` method:

```
Membership.UpdateUser(userInfo)
```

What happens when the `GetUser` and `UpdateUser` methods are called depends on the configured Membership provider. As discussed in Part 1 of this series, the Membership system is built using the provider model, meaning that the Membership API - the `Membership` class - simply defines a set of methods and properties. The actual work is handled by a configured provider. We've been using the `SqlMembershipProvider` provider throughout the course of this article series. As its name implies, the `SqlMembershipProvider` provider uses a SQL Server database as the user store. Consequently, when we call `GetUser` or `UpdateUser` from our code, the provider connects to the specified database and runs a stored procedure that gets information about the requested user or updates the specified user's account.

Ideally, anytime you need to work with the Membership system or examine or modify user account information you should use the `Membership` class and avoid going directly to the underlying database. Working directly with the database ties you to a specific provider implementation. Moreover, working directly with the database is more error-prone as the Membership API sets up a nice black box on your behalf, has straightforward named properties and methods, and is well documented. The database schema used by `SqlMembershipProvider` provider lacks any sort of documentation and will have you guessing as to what tables and columns store what data. **Unfortunately, idealism and pragmatism only rarely intersect. In some cases - such as allowing a user to change their username - we have no choice but to work directly with the underlying data store.**

### Another Example of Bypassing the Membership API

The `MembershipUser` class includes a method for changing a user's password. This method takes as input the old password and the new password. Furthermore, if the Membership system is configured to use security questions and answers (the default behavior) then this method must also be passed the security answer. This approach works great for users who want to change their password, as they know their old password and security answer. But what if you need to let an administrator change a user's password? The administrator doesn't know the old password or the security answer and therefore cannot change the user's password.

To circumvent these checks you need to go directly to the underlying user store and changing the password in the database, bypassing the Membership API. For more information on this particular scenario refer to the "Allowing Administrators to Change Users' Passwords" section in the Recovering and Changing Passwords tutorial, which is one of 14 tutorials in my Website Security Tutorials series.

## An Overview of How the `SqlMembershipProvider` Provider Stored User Account Information

The `SqlMembershipProvider` provider is designed so that one database can hold user accounts for several different applications. Each user account is associated with an *application*. Whenever the Membership system goes to the database to get information about a user or to update a user or to create a user it passes in the *application name* so that the user is retrieved/updated/created from the appropriate application. The applications that partition the users are maintained in a database table named `aspnet_Applications`; each application has an `ApplicationName` field, but is uniquely identified via the `ApplicationId` field.

User account information is stored in two tables:

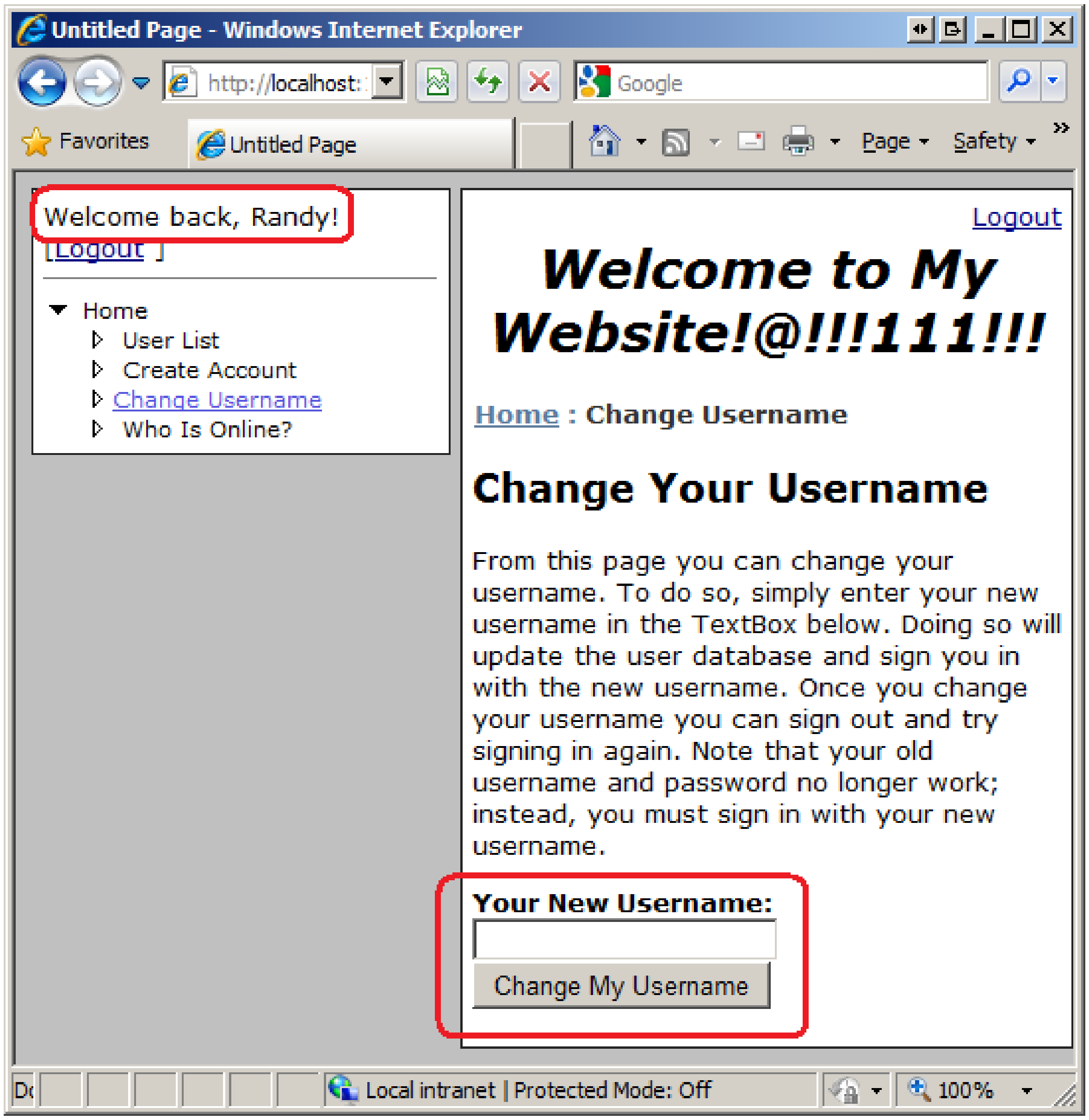
- `aspnet_Users` - contains a row for each user in the system. Each user is uniquely identified via the `UserId` property and each user has an `ApplicationId` value that indicates what application the user account belongs to. This table only stores the core user-related fields: `UserId`, `UserName`, and `LastActivityDate`. There's also a `LoweredUserName` field that stores the username but in all lowercase letters. This field is present for database configurations that are case sensitive.
- aspnet_Membership - stores additional user information, such as the user's password, e-mail address, whether or not they're approved, their last login date, and so forth.

In order to change a user's username we'll need to update the `UserName` and `LoweredUserName` fields in the `aspnet_Users` table.

**Creating a "Change Your Username" Page**

To illustrate changing a username I've created a sample application that includes a page named `ChangeUsername.aspx`. (This demo application is available for download at the end of this article.) To change his username a user would sign into the site with his current username and then visit `ChangeUsername.aspx` and enter the new username. From the end user's perspective, that's all there is to it. Things are a little more complicated for us, the page developer. First, we need to ensure that the new username the user is requesting is not already taken. Next, we must burrow down into the database and update the `UserName` and `LoweredUserName` fields in the `aspnet_Users` table. Finally, we need to "re-sign in" the user with their new username.

Before we jump into the code, let's first talk about this page's user interface. This page needs a TextBox control for the user to enter his new username and a Button control that, when clicked, will kick off the username change. The following screen shot shows this page when first visited by a user. I placed this page in the `UsersOnly` folder, which has a `Web.config` that locks down the contents of this folder to only authenticated users. Consequently, if an anonymous user visits this page they will be automatically redirected back to the login page. As you can see in the screen shot below, user Randy is currently signed in.



Welcome to My Website!@!!!111!!!

Home : Change Username

Change Your Username

From this page you can change your username. To do so, simply enter your new username in the TextBox below. Doing so will update the user database and sign you in with the new username. Once you change your username you can sign out and try signing in again. Note that your old username and password no longer work; instead, you must sign in with your new username.

Your New Username:

Change My Username

There are two additional Web controls in the `ChangeUsername.aspx` page that are not shown in the above screen shot. The first is a Label Web control named `lblErrors` and is used to display information about any errors that occur during the username renaming process, such as if the username cannot be changed because the desired new username is already used by someone else. There is also a RequiredFieldValidator control on the TextBox to ensure that the user enters a new username.

**Ensuring the New Username is Not Already Taken**

When a user enters a username and clicks the "Change My Username" Button we need to make sure that the desired username is not already in use. This is accomplished by calling the `Membership.GetUser` method and passing in the desired username. The `GetUser` method will return a `MembershipUser` object if the username is already taken, and `Nothing` (or `null`, in C#) otherwise. This logic is implemented in the `btnChangeUsername` Button's `Click` e

```
Protected Sub btnChangeUsername_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles btnChangeUsername.Click
   ...

   'Does this username already exist?
   Dim newUsername As String = txtUsername.Text.Trim()
   Dim usr As MembershipUser = Membership.GetUser(newUsername)
   If usr IsNot Nothing Then
      lblErrors.Text = String.Format("The username {0} is already being used by someone
else. Please try entering a different username.", newUsername)
      Exit Sub
   End If

   ...
```

Note that if the username is already taken then the `lblErrors` Label's `Text` property is set to an appropriate error string and control exits from the event handler.

## Changing the Username

As aforementioned, changing the username requires that we work directly with the underlying database to update the `UserName` and `LoweredUserName` fields in the `aspnet_Users` table. I created a stored procedure named `usp_ChangeUsername` that performs this logic. The `usp_ChangeUsername` stored procedure accepts three input parameters:

- `@ApplicationName` - the name of the application. This information is available programmatically via the `Membership.ApplicationName` property.
- `@OldUserName` - the user's current username
- `@NewUserName` - the user's desired new username

This stored procedure looks up the `UserId` and `ApplicationId` values for the user from the `aspnet_Applications` and `aspnet_Users` tables. If no user is found in the database then the stored procedure exits with a return code of 1. Next, a final check is done to ensure that the username is not already taken. If it is taken then the stored procedure exits with a return code of 2. If everything checks out, the username is updated and the stored procedure returns a value of 0.

```
-- Changes the username for @OldUserName to @NewUserName (in application
@ApplicationName)
-- Returns:
-- 0 if success
-- 1 if @OldUserName not found
-- 2 if @NewUserName is already taken

ALTER PROCEDURE dbo.usp_ChangeUsername
   @ApplicationName    nvarchar(256),
   @OldUserName       nvarchar(256),
   @NewUserName       nvarchar(256)
AS
BEGIN

-- Get the UserId and ApplicationId for the user
   DECLARE @UserId uniqueidentifier, @ApplicationId uniqueidentifier
   SELECT @UserId = NULL

   SELECT @UserId = u.UserId, @ApplicationId = a.ApplicationId
   FROM   dbo.aspnet_Users u, dbo.aspnet_Applications a
   WHERE LoweredUserName = LOWER(@OldUserName) AND
         u.ApplicationId = a.ApplicationId AND
         LOWER(@ApplicationName) = a.LoweredApplicationName

   IF (@UserId IS NULL)
      RETURN(1)


   -- Ensure that @NewUserName is not in use
   IF (EXISTS(SELECT 1 FROM aspnet_Users WHERE LoweredUserName = LOWER(@NewUserName) AND
ApplicationId = @ApplicationId))
      RETURN(2)


   -- Change the username
   UPDATE aspnet_Users SET
        UserName = @NewUserName,
        LoweredUserName = LOWER(@NewUserName)
   WHERE UserId = @UserId AND ApplicationId = @ApplicationId

   RETURN(0)
END
```

This stored procedure is called from a method named `ChangeUsername` in a helper class named `UserAPI.vb`, which you'll find in the `App_Code` folder of the download at the end of this article. The `ChangeUsername` method uses ADO.NET to connect to the database and execute the stored procedure, passing in values for the three parameters and creating a parameter for the return value. The method returns True if the username was successfully changed, False otherwise.

```
Public Shared Function ChangeUsername(ByVal oldUsername As String, ByVal newUsername As
String) As Boolean
   Using myConnection As New SqlConnection()
      myConnection.ConnectionString =
ConfigurationManager.ConnectionStrings("MembershipConnectionString").ConnectionString

      Dim myCommand As New SqlCommand
      myCommand.Connection = myConnection
      myCommand.CommandText = "usp_ChangeUsername"
      myCommand.CommandType = CommandType.StoredProcedure

      myCommand.Parameters.Add(CreateInputParam("@ApplicationName", SqlDbType.NVarChar,
Membership.ApplicationName))
      myCommand.Parameters.Add(CreateInputParam("@OldUserName", SqlDbType.NVarChar,
oldUsername))
      myCommand.Parameters.Add(CreateInputParam("@NewUserName", SqlDbType.NVarChar,
newUsername))

      Dim retValParam As New SqlParameter("@ReturnValue", SqlDbType.Int)
      retValParam.Direction = ParameterDirection.ReturnValue
      myCommand.Parameters.Add(retValParam)

      myConnection.Open()
      myCommand.ExecuteNonQuery()
      myConnection.Close()

      Dim returnValue As Integer = -1
      If retValParam.Value IsNot Nothing Then
         returnValue = Convert.ToInt32(retValParam.Value)
      End If

      If returnValue <> 0 Then
         Return False
      Else
         Return True
      End If
   End Using
End Function
```

This method is called from the `ChangeUsername.aspx` web page with the following code:

```
Protected Sub btnChangeUsername_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles btnChangeUsername.Click
   ...

   Dim success As Boolean = UserAPI.ChangeUsername(User.Identity.Name, newUsername)

   ...
```

`User.Identity.Name` returns the username of the currently logged in user. `newUsername` is a variable that was declared earlier in the `Click` event handler and holds the value the user entered into the `txtUsername` TextBox.

**Re-Signing the User In With His New Username**

If the username is successfully updated we have one final step: we need to "re-sign in" the user. When a user signs in to a website a forms authentication ticket is created on their browser. This ticket is a token that identifies the user and includes their username, among other information. We need to update this authentication ticket to store the new username, otherwise the user will continue to see messages like, "Welcome back, *OldUserName*" until they sign out and re-sign in manually.

The good news is that it's remarkably easy to create the authentication ticket and specify the username. The .NET Framework includes a `FormsAuthentication` class which has a method named `SetAuthCookie`. This method accepts two input parameters: the user's name and whether to stored the authentication in a persistent cookie. (A persistent cookie is one that survives browser restarts, whereas a non-persistent one is trashed whenever the user closes their browser.)

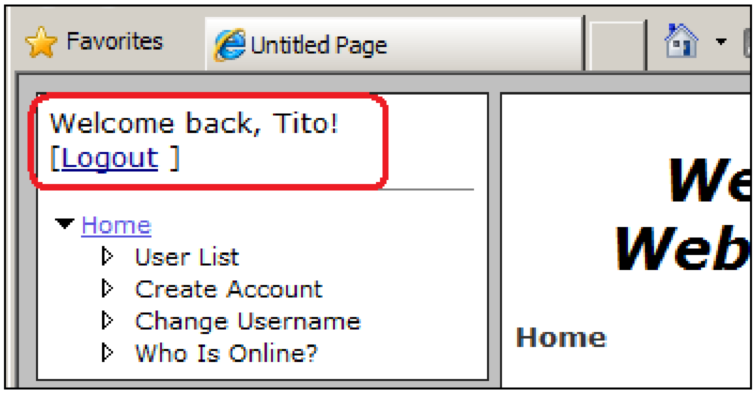
The following code re-signs in the user by creating a new non-persistent authentication ticket with the new username. The user is then redirected to the site's homepage, `Default.aspx`. Keep in mind that this code only runs if the call to `UserAPI.ChangeUsername` returned True. If the `ChangeUsername` method returned False then there was some problem in changing the user's name and a message is displayed indicating such.

```
Protected Sub btnChangeUsername_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles btnChangeUsername.Click<br />
   ...

   If success Then
      'Sign in user using new username & send them back to homepage
      FormsAuthentication.SetAuthCookie(newUsername, False)

      Response.Redirect("~/Default.aspx")
   Else
      lblErrors.Text = "There were problems. We were unable to change your username."
   End If
End Sub
```

The screen shot below shows the homepage after Randy has changed his username from Randy to Tito. Note that the "Welcome back" message shows the user's new username, Tito.



**Important Considerations...**
The technique for changing a username described in this article is sufficient for scenarios where changing the username requires only a change to the `UserName` and `LoweredUserName` fields in the `aspnet_Users` table. However, more database changes may be needed. In the Introduction I noted that the ability for users to change their username is paramount for sites that use an e-mail address as the username. If you are using an e-mail address as the username chances are you may be storing the e-mail address the user enters when creating an account in two places: in the `aspnet_Users.UserName` field (and, by extension, in `LoweredUserName`) and in the `aspnet_Membership.Email` field. In that case, updating a user's username would entail also updating the `aspnet_Membership.Email` field.

Many web applications that support user accounts have database tables that need to "point back" to a user record. This is commonly done by having a field in the related table named `UserId` that serves as a foreign key constraint back to `aspnet_Users.UserId`. For example, consider an online messageboard site. This would have the need to support user accounts and could use the Membership system. It would also have a table named `Posts` that would have a record for each posting to the messageboard. In order to determine which users made which posts we'd need to add a column to the `Posts` table that links a post to a user. Ideally, this relationship would be reflected by adding a `Posts.UserId` field of type `uniqueidentifier` and creating a foreign key constraint between that column (the foreign key) and `aspnet_Users.UserId`, the primary key.

However, I have seen plenty of database schemas where the database designer linked back on the `aspnet_Users.UserName` field instead of `UserId`. Returning to the Posts example, that would mean that there would not be a `Posts.UserId` field, but instead a `Posts.UserName` field. If you used this approach then allowing the user to change their username adds a bit of extra work because now when a user changes his username you also need to update each record in the `Posts` table, replacing the old `UserName` value with the new one.

Both of these tasks - updating `aspnet_Membership.Email` and updating `UserName` columns in related tables - can be done by adding one or more `UPDATE` statements to the `usp_ChangeUsername` stored procedure.

Happy Programming!

- By Scott Mitchell

---

## Further Reading

- Website Security Tutorials (VB & C# Versions Available)
- Recovering and Changing Passwords (VB Version) (C# Version)

## Attachments

- Download the code used in this article

| Article Information | |
|---|---|
| Article Title: | ASP.NET.Examining ASP.NET's Membership, Roles, and Profile - Part 15 |
| Article Author: | Scott Mitchell |
| Published Date: | July 1, 2009 |
| Article URL: | http://www.4GuysFromRolla.com/articles/070109-1.aspx |